Try GX Core

Start here to learn how to connect to data, create Expectations, validate data, and review Validation Results. This is an ideal place to start if you're new to GX Core and want to experiment with features and see what it offers.

To complement your code exploration, check out the GX Core overview for a primer on the GX Core components and workflow pattern used in the examples.

Prerequisites

- Python version 3.9 to 3.12

Setup

GX Core is a Python library you can install with the Python `pip` tool.

For more comprehensive guidance on setting up a Python environment, installing GX Core, and installing additional dependencies for specific data formats and storage environments, see Set up a GX environment.

1. Run the following terminal command to install the GX Core library:

   Terminal input
   ```
   pip install great_expectations
   ```

2. Verify GX Core installed successfully by running the command below in your Python interpreter, IDE, notebook, or script:

   Python input
   ```python
   import great_expectations as gx

   print(gx.__version__)
   ```

   If GX was installed correctly, the version number of the installed GX library will be printed.

Sample data

The examples provided on this page use a sample of NYC taxi trip record data. The sample data is provided using multiple mediums (CSV file, Postgres table) to support each workflow.

When using the taxi data, you can make certain assumptions. For example:

- The passenger count should be greater than zero because at least one passenger needs to be present for a ride. And, taxis can accommodate a maximum of six passengers.
- Trip fares should be greater than zero.

Validate data in a DataFrame

This example workflow walks you through connecting to data in a Pandas DataFrame and validating the data using a single Expectation.

Pandas install
This example requires that Pandas is installed in the same Python environment where you are running GX Core.

Procedure

Instructions | Sample code

Run the following steps in a Python interpreter, IDE, notebook, or script.

1. Import the `great_expectations` library.

   The `great_expectations` module is the root of the GX Core library and contains shortcuts and convenience methods for starting a GX project in a Python session.

   The `pandas` library is used to ingest sample data for this example.

   Python input
   ```python
   import great_expectations as gx

   import pandas as pd
   ```

2. Download and read the sample data into a Pandas DataFrame.

   Python input
   ```python
   df = pd.read_csv(
       "https://raw.githubusercontent.com/great-
   expectations/gx_tutorials/main/data/yellow_tripdata_sample_2019-01.csv"
   )
   ```

3. Create a Data Context.

   A Data Context object serves as the entrypoint for interacting with GX components.

   Python input
   ```python
   context = gx.get_context()
   ```

4. Connect to data and create a Batch.

   Define a Data Source, Data Asset, Batch Definition, and Batch. The Pandas DataFrame is provided to the Batch Definition at runtime to create the Batch.

   Python input
   ```python
   data_source = context.data_sources.add_pandas("pandas")
   data_asset = data_source.add_dataframe_asset(name="pd dataframe asset")

   batch_definition = data_asset.add_batch_definition_whole_dataframe("batch definition")
   batch = batch_definition.get_batch(batch_parameters={"dataframe": df})
   ```

5. Create an Expectation.

   Expectations are a fundamental component of GX. They allow you to explicitly define the state to which your data should conform.

   Run the following code to define an Expectation that the contents of the column `passenger_count` consist of values ranging from `1` to `6`:

   Python input
   ```python
   expectation = gx.expectations.ExpectColumnValuesToBeBetween(
       column="passenger_count", min_value=1, max_value=6
   )
   ```

6. Run the following code to validate the sample data against your Expectation and view the results:

   Python input
   ```python
   validation_result = batch.validate(expectation)
   print(validation_result)
   ```

   The sample data conforms to the defined Expectation and the following Validation Results are returned:

   Python output
   ```
   {
     "success": true,
     "expectation_config": {
       "type": "expect_column_values_to_be_between",
       "kwargs": {
         "batch_id": "pandas-pd dataframe asset",
         "column": "passenger_count",
         "min_value": 1.0,
         "max_value": 6.0
       },
       "meta": {}
     },
     "result": {
       "element_count": 10000,
       "unexpected_count": 0,
       "unexpected_percent": 0.0,
       "partial_unexpected_list": [],
       "missing_count": 0,
       "missing_percent": 0.0,
       "unexpected_percent_total": 0.0,
       "unexpected_percent_nonmissing": 0.0,
       "partial_unexpected_counts": [],
       "partial_unexpected_index_list": []
     },
     "meta": {},
     "exception_info": {
       "raised_exception": false,
       "exception_traceback": null,
       "exception_message": null
     }
   }
   ```

Validate data in a SQL table

This example workflow walks you through connecting to data in a Postgres table, creating an Expectation Suite, and setting up a Checkpoint to validate the data.

Procedure

Instructions | Sample code

Run the following steps in a Python interpreter, IDE, notebook, or script.

1. Import the `great_expectations` library.

   The `great_expectations` module is the root of the GX Core library and contains shortcuts and convenience methods for starting a GX project in a Python session.

   Python input
   ```python
   import great_expectations as gx
   ```

2. Create a Data Context.

   A Data Context object serves as the entrypoint for interacting with GX components.

   Python input
   ```python
   context = gx.get_context()
   ```

3. Connect to data and create a Batch.

   Define a Data Source, Data Asset, Batch Definition, and Batch. The connection string is used by the Data Source to connect to the cloud Postgres database hosting the sample data.

   Python input
   ```python
   connection_string =
   "postgresql+psycopg2://try_gx:try_gx@postgres.workshops.greatexpectations.io/gx_example_db"

   data_source = context.data_sources.add_postgres(
       "postgres db", connection_string=connection_string
   )
   data_asset = data_source.add_table_asset(name="taxi data", table_name="nyc_taxi_data")

   batch_definition = data_asset.add_batch_definition_whole_table("batch definition")
   batch = batch_definition.get_batch()
   ```

4. Create an Expectation Suite.

   Expectations are a fundamental component of GX. They allow you to explicitly define the state to which your data should conform. Expectation Suites are collections of Expectations.

   Run the following code to define an Expectation Suite containing two Expectations. The first Expectation expects that the column `passenger_count` consists of values ranging from `1` to `6`, and the second expects that the column `fare_amount` contains non-negative values.

   Python input
   ```python
   suite = context.suites.add(
       gx.core.expectation_suite.ExpectationSuite(name="expectations")
   )
   suite.add_expectation(
       gx.expectations.ExpectColumnValuesToBeBetween(
           column="passenger_count", min_value=1, max_value=6
       )
   )
   suite.add_expectation(
       gx.expectations.ExpectColumnValuesToBeBetween(column="fare_amount", min_value=0)
   )
   ```

5. Create an Validation Definition.

   The Validation Definition explicitly ties together the Batch of data to be validated to the Expectation Suite used to validate the data.

   Python input
   ```python
   validation_definition = context.validation_definitions.add(
       gx.core.validation_definition.ValidationDefinition(
           name="validation definition",
           data=batch_definition,
           suite=suite,
       )
   )
   ```

6. Create and run a Checkpoint to validate the data based on the supplied Validation Definition. `.describe()` is a convenience method to view a summary of the Checkpoint results.

   Python input
   ```python
   checkpoint = context.checkpoints.add(
       gx.checkpoint.checkpoint.Checkpoint(
           name="checkpoint", validation_definitions=[validation_definition]
       )
   )

   checkpoint_result = checkpoint.run()
   print(checkpoint_result.describe())
   ```

   The returned results reflect the passing of one Expectation and the failure of one Expectation.

   When an Expectation fails, the Validation Results of the failed Expectation include metrics to help you assess the severity of the issue:

   Python input
   ```
   {
       "success": false,
       "statistics": {
           "evaluated_validations": 1,
           "success_percent": 0.0,
           "successful_validations": 0,
           "unsuccessful_validations": 1
       },
       "validation_results": [
           {
               "success": false,
               "statistics": {
                   "evaluated_expectations": 2,
                   "successful_expectations": 1,
                   "unsuccessful_expectations": 1,
                   "success_percent": 50.0
               },
               "expectations": [
                   {
                       "expectation_type": "expect_column_values_to_be_between",
                       "success": true,
                       "kwargs": {
                           "batch_id": "postgres db-taxi data",
                           "column": "passenger_count",
                           "min_value": 1.0,
                           "max_value": 6.0
                       },
                       "result": {
                           "element_count": 20000,
                           "unexpected_count": 0,
                           "unexpected_percent": 0.0,
                           "partial_unexpected_list": [],
                           "missing_count": 0,
                           "missing_percent": 0.0,
                           "unexpected_percent_total": 0.0,
                           "unexpected_percent_nonmissing": 0.0,
                           "partial_unexpected_counts": []
                       }
                   },
                   {
                       "expectation_type": "expect_column_values_to_be_between",
                       "success": false,
                       "kwargs": {
                           "batch_id": "postgres db-taxi data",
                           "column": "fare_amount",
                           "min_value": 0.0
                       },
                       "result": {
                           "element_count": 20000,
                           "unexpected_count": 14,
                           "unexpected_percent": 0.06999999999999999,
                           "partial_unexpected_list": [
                               -0.01,
                               -52.0,
                               -0.1,
                               -5.5,
                               -3.0,
                               -52.0,
                               -4.0,
                               -0.01,
                               -52.0,
                               -0.1,
                               -5.5,
                               -3.0,
                               -52.0,
                               -4.0
                           ],
                           "missing_count": 0,
                           "missing_percent": 0.0,
                           "unexpected_percent_total": 0.06999999999999999,
                           "unexpected_percent_nonmissing": 0.06999999999999999,
                           "partial_unexpected_counts": [
                               {
                                   "value": -52.0,
                                   "count": 4
                               },
                               {
                                   "value": -5.5,
                                   "count": 2
                               },
                               {
                                   "value": -4.0,
                                   "count": 2
                               },
                               {
                                   "value": -3.0,
                                   "count": 2
                               },
                               {
                                   "value": -0.1,
                                   "count": 2
                               },
                               {
                                   "value": -0.01,
                                   "count": 2
                               }
                           ]
                       }
                   }
               ],
               "result_url": null
           }
       ]
   }
   ```

   To reduce the size of the results and make it easier to review, only a portion of the failed values and record indexes are included in the Checkpoint results. The failed counts and percentages correspond to the failed records in the validated data.

Next steps

- Go to the Expectations Gallery and experiment with other Expectations.
- If you're ready to start using GX Core with your own data, the Set up a GX environment documentation provides a more comprehensive guide to setting up GX to work with specific data formats and environments.
- Check out GX Cloud, our SaaS platform—it's now in public preview! Sign up here and you could be validating your data in minutes. We also offer regular GX Cloud workshops: click here to get more information and register.

Was this topic helpful? Yes No